# sig.py

someonesdad1@gmail.com 31 Jan 2013
Updated 27 May 2013

The `sig.zip` package contains the `sig.py` python script and this document, `sig.pdf`. `sig.py` is a module used to:

1. Format numbers to a given number of significant figures.
2. Interpret strings to yield a number or a number with an uncertainty. Both can also contain an optional string representing a physical unit.

As `sig.py` will primarily be of interest to python programmers, I'll assume you have python and will know where to install the `sig.py` script to meet your needs.

## Why would you want to use it?

For many applications, the tools built into python can format a number suitably for general-purpose use. For example, C-type string interpolation can be used with the `g` formatting option to approximate getting a desired number of significant figures (example: `"%.2g"` will give 2 significant figures). Or the string formatting tool `''.format(...)` can be used to do similar things.

If these tools work for you, use them, as they're simple and robust.

However, one objection to them is that you have little to no control over when the formatting decides to switch to scientific notation. This is a flaw in some applications where you're trying to e.g. format things in columns for a nice ASCII display. Over the last 10-15 years, I've written a number of formatting tools in python, C, and C++ and was finally led to the realization that I should write a formatting tool for python that would conveniently format numbers the way I wanted them to be formatted (over the last decade or so, I've moved to using python for virtually all of my applications). Over the years, the features of this formatting tool grew to the present list:

1. Will format integers, python floats, python complex numbers, python Decimal numbers, python fractions, mpmath floating point and complex numbers, and `ufloat` numbers from the python `uncertainties` module. It will also format strings that represent a number or a group of numbers.
2. The module is capable of formatting a sequence of the above numbers, including numpy arrays. However, it currently doesn't arrange numpy array output as nicely as numpy does (that's on the to-do list for the future).
3. There are numerous attributes to the formatting object (`SigFig`) that allow detailed control over how a number is formatted because humans like to write numbers in quite a few different ways.
4. The module will make use of the current locale if appropriate, but will not change the current locale.
5. A feature (`AlignDP()`) can adjust the number of significant figures in the output to get it to fit in a specified number of spaces. This is useful for tabular data where the decimal points line up.
6. Since I write numerous programs that get information from the user, process it, then print out the results, I also wanted a general-purpose routine that would interpret a number with an optional uncertainty and physical units. Thus, the `sig.py` module has a `sig.Interpret()` method that can do this. This allows me to write many applications that need only one library module that isn't part of python's standard library.

It's possible there are still subtle errors in this module -- writing code to format numbers as humans like to see them is more difficult and involved than most people think. If you find a bug, please send it to me at the email address in the title; I'll fix the code and add it to the test cases.

As of 14 Aug 2014, the `sig.py` module has been tested with python 2.7.6, and 3.4.0. Interestingly, the self-tests run about 5 times faster under python 3.4.0 than they do under 2.7.6.

Let's show some examples of use to give you a better idea of some of the features.

## Examples

One of the motivations for the `sig.py` module was that python doesn't do a good job of formatting numbers from a usability perspective. Here's an example; suppose I have the array

```
x = ["2-1/3", (22/7., -3+4.1j)]
```

and I call `print(x)` to get it printed to stdout:

```
['2-1/3', (3.1428571428571428, (-3+4.0999999999999996j))]
```

What you see will depend on the python version you're using. This isn't too bad for a small array, but if you're printing lots of numbers, you can be annoyed or overwhelmed by the extraneous digits.

Using `print(sig(x))` results in (note the fraction was converted to a float)

```
[2.33, (3.14, -3.00+4.10i)]
```

Now try this example

```
a = []
for i in range(1, 50):
    a.append(1./i)
print(a)
```

and you'll see a mess; here's what was on my console's screen:

```
[1.0, 0.5, 0.3333333333333333, 0.25, 0.2, 0.16666666666666666,
0.14285714285714285, 0.125, 0.1111111111111111, 0.1, 0.09090909090909091,
0.08333333333333333, 0.07692307692307693, 0.07142857142857142,
0.06666666666666667, 0.0625, 0.058823529411764705, 0.05555555555555555,
0.05263157894736842, 0.05, 0.047619047619047616, 0.045454545454545456,
0.043478260869565216, 0.041666666666666664, 0.04, 0.038461538461538464,
0.037037037037037035, 0.03571428571428571, 0.034482758620689655,
0.03333333333333333, 0.03225806451612903, 0.03125, 0.030303030303030304,
0.029411764705882353, 0.02857142857142857, 0.027777777777777776,
0.02702702702702703, 0.02631578947368421, 0.02564102564102564, 0.025,
0.024390243902439025, 0.023809523809523808, 0.023255813953488372,
0.022727272727272728, 0.022222222222222223, 0.021739130434782608,
0.02127659574468085, 0.020833333333333332, 0.02040816326530612]
```

If instead we used

```
from sig import sig
a = []
for i in range(1, 50):
    a.append(1./i)
print(sig(a))
```

you'd get the easier-to-read

```
[1.00, 0.500, 0.333, 0.250, 0.200, 0.167, 0.143, 0.125, 0.111, 0.100, 0.0909,
0.0833, 0.0769, 0.0714, 0.0667, 0.0625, 0.0588, 0.0556, 0.0526, 0.0500,
0.0476, 0.0455, 0.0435, 0.0417, 0.0400, 0.0385, 0.0370, 0.0357, 0.0345,
0.0333, 0.0323, 0.0312, 0.0303, 0.0294, 0.0286, 0.0278, 0.0270, 0.0263,
0.0256, 0.0250, 0.0244, 0.0238, 0.0233, 0.0227, 0.0222, 0.0217, 0.0213,
0.0208, 0.0204]
```

This is actually printed on one line, so what you're seeing in both cases includes wrapping to screen width. However, the point is that limiting things to three significant figures (the default) makes things easier to understand. This is the fundamental purpose of the `sig.py` module and, in fact, python's "noisy" printing of floating point arrays is the canonical use case that led me to develop this module.

Uncertain values are printed with short-hand notation by default:

```
from uncertainties import ufloat
from sig import sig
x = ufloat(1.2345, 0.00678)
print(sig(x))
sig.unc_short = False
print(sig(x))
```

produces

```
1.234(7)
1.23+/-0.007
```

The second form uses the default 3 significant figures for the significand and the default 1 significant figure for the uncertainty.

# Usage

The main class in the `sig.py` module is the `SigFig` object. You can define instances of this class and call them with numbers to get the desired behavior. You can customize the behavior by setting the attributes.

Because a single instance of a `SigFig` object is often all that's needed for a script, the `sig` object is a convenience instance of the `SigFig` class. This is the object I use for all my formatting needs, as it's simple to type into a script

```
from sig import sig
```

then make calls like `sig(x)` to format a number or sequence `x`.

One disadvantage of using `sig` is that the attribute changes are "global" (because `sig` is in reality a single class instance). This means if you change an attribute in one part of a program, that change will also be seen in another part. If you want to avoid this, define local instances of the `SigFig` object.

# SigFig object's attributes

The conversion of a number to a string is controlled by the `SigFig` object's attributes. Most of these attributes are not used with getters/setters, so you can change them to values that may cause exceptions or undesired output (in which case I'd say "don't do that"). If you start getting exceptions, you can call the `sig.check()` method which should raise an exception on an incorrectly-set attribute. A few of these attributes do have getters/setters because they must be of a specific type internally.

You can edit the `SigFig` object's variables to change the default values of the attributes. There are a surprising number of attributes because there are quite a few ways to format floating point and complex numbers.

In the following, the context is the call `sig(x, digits)`. Again, `sig` is a convenience instance of the `SigFig` object.

## digits

This attribute can be an integer, float, Decimal, Fraction, or string. If it's an integer, it controls the number of significant figures in the output string. You will get accurate results for Decimal and mpmath numbers, but for the platform's floating point numbers (like used in floats and in numpy), you will get meaningless trailing digits if you exceed the floating point precision.

Non-integer types cause `digits` to be used as a rounding template. The algorithm used for rounding `x` to the template `digits` is (sign is the sign of `x`, either +1 or -1):

```
str(sign*int(abs(x)/digits + Decimal("0.5"))*digits)
```

In other words, find out how many integer units of `digits` are in the number `x`, then multiply that integer by `digits.` If you'd prefer a more sophisticated rounding algorithm, the method to change is `SigFig._TemplateRound`.

When you call `sig` via `sig(x, digits)`, the `digits` parameter overrides the current `sig.digits` setting. The intent is to allow you to set `sig.digits` once at the beginning of your code and get uniform printouts following. Occasionally, you might want a different number of significant figures without disturbing the default; the second parameter is for this use case.

## dp

This string is used to represent the decimal point. The default is set from the locale.

## dp_position

When nonzero, this integer is used to determine the location of the decimal point from the left edge of the width defined by the `fit` attribute. It is used to align decimal points at a stated number of spaces from the left edge of the space defined by `fit`. For this to happen, both `fit` and `dp_position` must be nonzero and `fit` must be larger than `dp_position`. Note this number is 0-based.

You can get a fitted number in two ways. Set the attributes `fit` and `dp_position` to nonzero values and subsequent calls to `sig()` will do the fitting. If you only want a few such fitted numbers, you can instead call the `sig.AlignDP()` method with `width` keyword parameter for the `fit` attribute and `position` keyword parameter for the `dp_position` attribute.

## echar

Sets the character (or string) that you want to use for indicating an exponent. Defaults to `e`.

## edigits

The minimum number of digits in the exponent; unused leading digits are zero.

## esign

If `True`, include the `+` sign for a positive exponent.

## fit

If nonzero, the integer `digits` is adjusted downward until the returned string consumes no more than `abs(fit)` spaces. If `fit` is positive, the string is right-justified; if negative, left justified. If the string cannot be fit in the desired space, the string `None` is returned (and it will be truncated if necessary to fit in the indicated number of spaces). This option works well with a table-printing tool like `texttable.py` (see [http://foutaise.org/code/](http://foutaise.org/code/)).

## high

If the number `x` has an absolute value larger than this number, then scientific notation is used.

## idp

If `True`, then numbers that look like integers will have a trailing decimal point. The intent is that this trailing decimal point indicates the number is a floating point number, not an integer.

## ignore_none

If `False` and if the argument `x` is `None`, then a `ValueError` exception is raised. If `True`, then

`None` is interpreted as zero.

### integer

If 0, an integer is converted to a floating point number and formatted as is normal. If 1, the integer is left as an integer and its `str()` value is returned. If > 1, then the same thing is done except it is formatted in the current locale. This means, for example, that `123456` would be returned as `123,456` in the US.

### lead_zero

If `True`, the leading zero for numbers `abs(x)` < 1 is included.

### low

If the number has an absolute value less than this number, then scientific notation is used.

### mixed

If you use template rounding with a fractional template, you'll get back a proper fraction if `mixed` is `True` and an improper fraction if `mixed` is `False`.

### rtz

If `True`, remove trailing zeros from the string to be returned. While this "defeats" the significant figure reporting feature, there are times when the feature is convenient.

### separator

String that is used to separate the numbers when formatting sequences.

### sign

If `True`, the sign of the number is always shown. If `False`, the sign is only shown for negative numbers.

### zero_limit

If this number is not zero, then an `x` argument will be interpreted as zero if `abs(x) < zero_limit`. This also applies to the real part of a complex `x`.

## Attributes for complex numbers

The following attributes apply to the formatting of complex numbers (the above attributes will apply to the individual real and imaginary parts):

### imag_before

If `True`, then the imaginary unit is put before the imaginary part of a complex number. Example: the complex number `3-4i` is represented as `3-i4` if `imag_before` is `True`.

### imag_deg

If the polar display for complex numbers is used, the angular part will be displayed in decimal degrees if `imag_deg` is `True`. Otherwise, radians are used.

## imag_deg_sym

The string used to represent degrees in the polar representation.

## imag_limit

An analogous attribute to `zero_limit`, but applied to the imaginary part of a complex number.

## imag_polar

If `True`, display a complex number in its polar representation. If `False`, use the rectangular representation.

## imag_polar_sep

String used to separate the magnitude and argument in a polar representation.

## imag_post

In a complex number, the string that comes after the sign combining the real and imaginary parts in the rectangular form.

## imag_pre

In a complex number, the string that comes before the sign combining the real and imaginary parts in the rectangular form.

## imag_sep

The string that separates the imaginary unit from the number's imaginary part.

## imag_unit

The string used to represent the imaginary unit.

# Attributes for displaying complex numbers as pairs of numbers

The following four attributes are used to show complex numbers as pairs of numbers. The conventional display for this is `(x, y)`, but this could be confused with a tuple. These four attributes let you choose how a pair display looks.

## imag_pair

If `True`, then display in pair form.

## imag_pair_left

The string for the left part of the pair.

## imag_pair_right

The string for the right part of the pair.

## imag_pair_sep

The string separating the real and imaginary components.

Example: if `imag_pair` is `True` and `imag_pair_left = "Complex("`, `imag_pair_right = ")"`, and `imag_pair_sep = ", "`, the complex number `1.234 + 5.678i` will be displayed as

`Complex(1.2, 5.7)` if `digits == 2`.

# Attributes for numbers with uncertainty

If you've allowed use of the `uncertainties` module, there are two forms of uncertainties that are used: the short form and the long form. The following attributes are used with `uncertainties.ufloat` numbers:

## unc_short

If `True`, uncertainties are expressed in the usual short-hand form of science. For example, if the value was `1.23456` and the uncertainty was `0.0026`, the short form is `1.234(3)`. For the short form, using `sig(x, digits)` causes the `digits` parameter to override the `sig.unc_digits` attribute. Thus,

```
sig(ufloat((1.23456, 0.0026)), 2)
```

returns `1.2346(26)`.

If `False`, the long form is used; the number of significant figures in the significand is controlled by the `digits` attribute and the `sig.unc_digits` attribute controls the number of digits in the uncertainty. For the given example, the long form would be expressed (where `digits` is `5` and `sig.unc_digits` is `2`) as `1.2346+/-0.0026`.

## unc_digits

Controls the number of significant figures in the uncertainty. If you call `sig(x, digits)`, the results depend on whether you're using the long form or the short form. For the long form, the `digits` argument always controls the number of digits in the significand. For the short form, the `digits` argument overrides the `sig.unc_digits` attribute and determines how many significant figures are displayed in the uncertainty; then the number of significant digits in the significand are determined by the uncertainty and how many digits it needs.

## unc_sep

Is the separation string between the value and its uncertainty in the long form.

## unc_pre

In the short form, this is the string that separates the value and its uncertainty.

## unc_post

In the short form, this is the string that follows the uncertainty.

# SigFig Methods

## AlignDP()

`AlignDP(num, width=None, position=None, digits=None)`

Fit the string into a stated width with the decimal point at the 0-based position starting from the left. The `width` parameter overrides the `self.fit` setting; `position` overrides the `self.dp_position` setting. Note the number significant figures in the result may be reduced to get it to fit into the given space.

The intent is to let you have a field of known width where the decimal points line up -- this is useful when displaying information that varies over a few orders of magnitude. The routine will display "-.-"

for numbers outside the displayable range.

**Warning**: this routine isn't fully debugged and still exhibits annoying behavior occasionally (I haven't gotten around to fixing it because I don't use it very often).

## check()

Check the attributes to ensure they are the proper types and values.

If you are getting an exception while using this module, run this method and it may point you to the cause of the problem.

## Interpret()

`Interpret(S, fp_type=float, glo=None, loc=None, strict=True)`

This method interprets the string `S` as a number or an assignment. Examples of allowed forms for `S` are

```
34 u            Integer
3.4u            Floating point
3.4[0.1]u       Mean 3.4 with uncertainty of 0.1
3.4+-0.1 u      Mean 3.4 with uncertainty of 0.1
3.4+/-0.1u      Mean 3.4 with uncertainty of 0.1
3.4(1) u        Short form uncertainty 3.4+-0.1
3 = 4           Assignment (strict = False), no unit
a = 4           Assignment (strict = True), no unit
a = b*c/d       Assignment (strict = True) w/ expr, no unit
b*c/d           Expression (no unit allowed)
```

All whitespace is removed from `S` before processing, so "number" forms like `"3 . 4"` or `"3. 4 ( 1 ) e- 4 m/s"` would be evaluated as expected.

The numbers can include an optional string `u` that will be interpreted as the physical units of the number. A number will first be interpreted as an integer; if that fails, then it will be interpreted as a floating point type of `fp_type`. Numbers with uncertainties will become `ufloats` from the `uncertainties` module if it is present; otherwise, the uncertainty is ignored. Note the units string `u` can have optional whitespace between it and the number part of the string.

The units string `u` should not contain any of the characters `()[]`; if it does, then this routine may fail to properly interpret the whole string. No units are allowed in assignments or expressions because they are evaluated by the python interpreter.

The method returns a tuple `(x, u)` where

1. `x` is an int, `u` is unit
2. `x` is an fp_type, `u` is unit
3. `x` is a ufloat, `u` is unit
4. `x` is assigned name, `u` is value
5. `x` is None, `u` is an error message

This isn't really a method needed by a significant figure object, but I included it because it captures the behaviors I need for getting input from users in my programs. The ability to define variables, evaluate python expressions, and define numbers with uncertainty covers virtually all of the cases I need for numerical input from a user (except for complex numbers). Thus, I use the `sig.py` module as a general-purpose input and output module. Often, the scripts I write don't need any other libraries that aren't built-in to python.

Note there's a "cost" associated with this routine: some things that are not valid numbers can be interpreted as numbers with units. For example, the string `3.4.4` will be interpreted as the floating point number `3.4` with an uncertainty string of `.4`.

`glo` is a dictionary used in subsequent expression evaluations as a global dictionary. It is not modified by the routine.

`loc` is a local dictionary use by the routine. If `loc` is not `None`, then if `S` is an assignment, the result is put into the `loc` dictionary.

## pop()

`pop(n=1)`

Removes and restores the last saved state. You can supply an integer argument `n` to specify how many saved states to pop or set `n` to `"all"` to pop all of them. It's not an error to try to `pop` more states than are on the stack.

## push()

When you use the `sig` convenience object, keeping track of which attribute is set can be painful -- and changing an attribute in one chunk of code is like modifying a global variable, as it can have effects in other code. There are two ways of dealing with this issue. One way is to instantiate the needed `SigFig` objects, then use and discard them locally.

Another way is to use the `push()` method to push the object's state onto an internal stack. You can then change the attributes as needed; when you're finished, call the `pop()` method and the `SigFig` object is returned to its former state. This feature is analogous to the push/pop methods of PostScript.

## reset()

Sets the `SigFig` object's attributes to their default values.

# Other sig.py module methods

## nsf()

`nsf(x, rrhz=False, dp=".")`

Return an integer counting the number of significant figures in the number represented by the string x. If `rrhz` is `True`, then also remove any right-hand zeros from the significand before counting. `dp` is the string used for the decimal point.

Note we allow short-hand uncertainty notation in `x` also, such as `1.234(5)e6` or `1.234[5]e6`.

## Examples()

If you run the `sig.py` script from the command line, you'll see some examples printed to stdout.